Lecture 4: Syntax Analysis

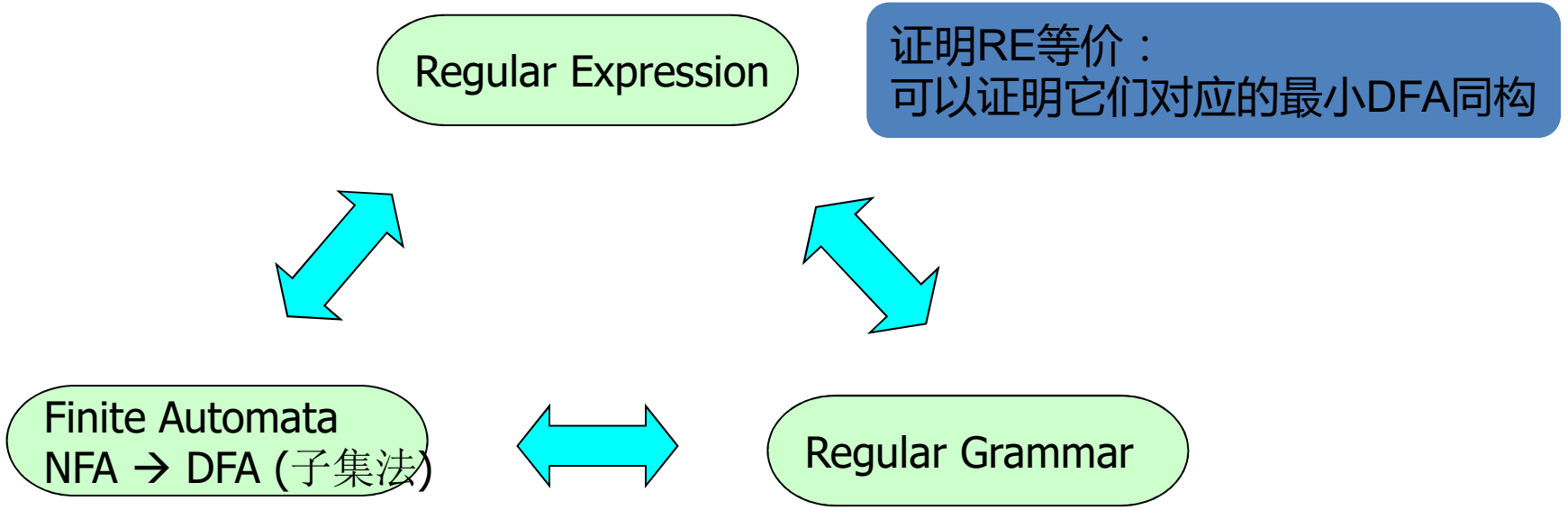
Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
计算机学院E301

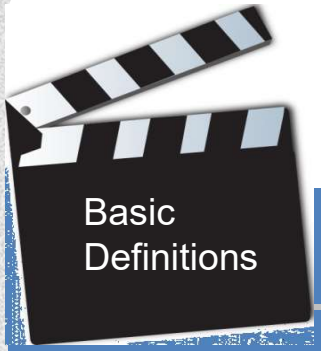
回顾

- **几个算法**
  - RE → NFA → DFA
  - RE → DFA
  - DFA最小化

3.7 RE v.s. NFA/DFA

- **RE，DFA(NFA)，L(RE)三者等价**

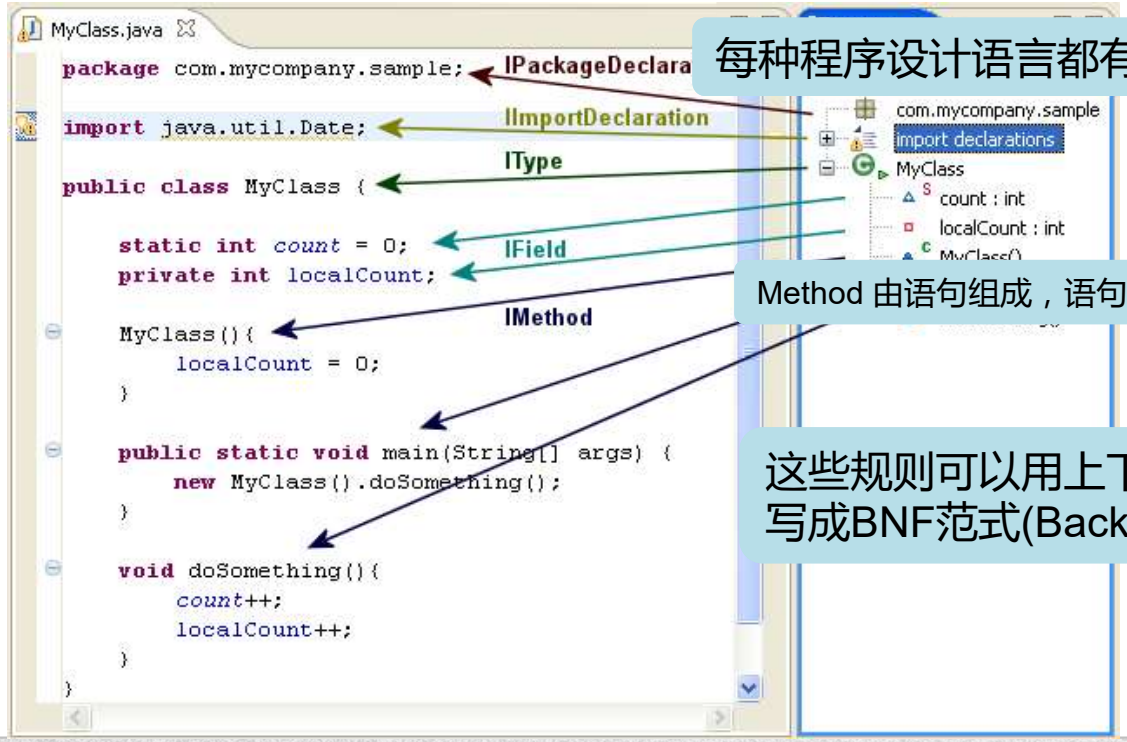




4.1 基本概念

4.1 基本概念

- 程序设计语言源程序的构成：语法结构



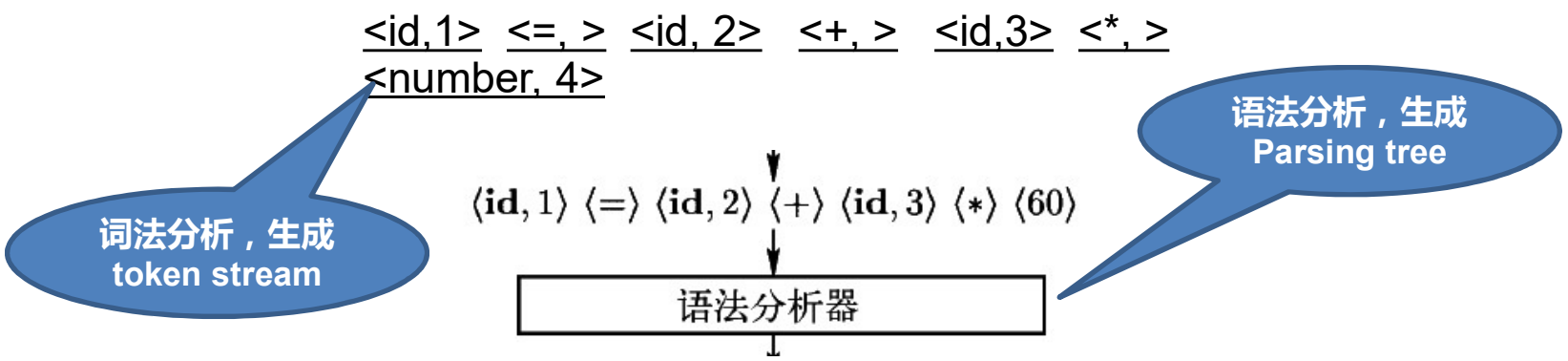
每种程序设计语言都有描述程序语法结构的规则。

Method 由语句组成，语句由表达式组成，表达式由记号组成

这些规则可以用上下文无关文法描述，PS：可以写成BNF范式(Backus-Naur From)

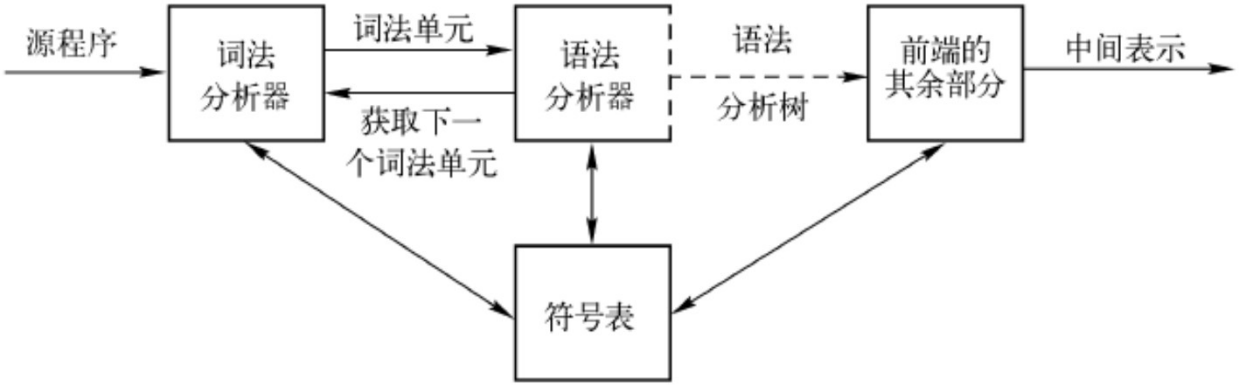
4.1 基本概念

- position = initial + rate * 60



4.1 基本概念

■ 语法分析器



类型检查，语义分析，翻译生成中间代码等往往和语法分析过程交错完成，实践中往往和语法分析放入一个模块，图上用“前端的其余部分”表示上述活动

4.1 基本概念

- **语法分析器**
  - 输入：词法分析器输出的词法单元序列
  - 输出：语法树表示
  - 功能：
    - 验证输入源程序的合法性，输出良构程序的语法结构
    - 对于病构的程序，能够报告语法错误，进行错误恢复
  - 类型：
    - 通用型
    - 自顶向下：通常处理LL文法
    - 自底向上：通常处理LR文法

4.2 上下文无关文法

4.2 上下文无关文法

- **Why Not Type-III Grammar (Regular Grammar, FSG)**
  - 正规文法能够做什么？
    - 描述词法
  - 正规文法不能做什么？
    - 描述配对结构，e.g. 由配对括号构成的串的
    - 语句的嵌套结构
    - 重复串$a^n b^n$

但这些都是语法结构所必须的

CFG can ☺

4.2 上下文无关文法

- 不是所有的Token串都是合法的程序，因此需要一个工具来描述合法的Token串。
- 一般地，程序设计语言的结构都是递归结构(recursive structure)
- 例如：表达式，

变量是表达式
（表达式）是表达式
表达式+表达式是表达式

- 上下文无关文法非常适合描述这种结构

4.2 上下文无关文法

- **文法(G, Grammar): 四元组G = (V_N,V_T, S, P)，其中**
  - V_N：一个非空有限的非终结符号集合，它的每个元素称为非终结符，一般用大写字母表示，它是可以被取代的符号；
  - V_T：一个非空有限的终结符号集合，它的每个元素称为终结符，一般用小写字母表示, 是一个语言不可再分的基本符号；
  - S：一个特殊的非终结符号，称为文法的开始符号或识别符号，$S \in V_N$。开始符号S必须至少在某个产生式的左部出现一次；
  - 设V是文法G的符号集，则有：$V = V_N \cup V_T$ ， $V_N \cap V_T = \emptyset$
  - P：产生式的有限集合。所谓的产生式，也称为产生规则或简称为规则，是按照一定格式书写的定义语法范畴的文法规则。

0,1,2,3型文法区别在于P

4.2 上下文无关文法

- **Chomsky 2型文法:上下文无关文法(CFG)**
  - P：$A \to \beta$，其中$A \in V_N$ ，$\beta \in V^*$。
  - 所有的产生式左边只有一个非终结符，产生式右部可以是V_N、V_T或ε
  - 非终结符的替换不必考虑上下文，故也称作上下文无关文法。

例：设文法$G_2=(\{S\},\{a,b\},P,S)$，其中P为:

(0) $S \to aSb$

(1) $S \to ab$

- **识别2型语言的自动机称为下推自动机(PDA)。**

4.2 上下文无关文法

符号表示约定

- 终结符

① 字母表中前面的小写字母，如 a，b，c；
② 黑体串，如 id 或 while；
③ 数字 0，1，…，9；
④ 标点符号，如括号，逗号等；
⑤ 运算符号，如＋，－等。

- 非终结符

① 字母表中前面的大写字母，如 A，B，C；
② 字母 *S*，*并且它通常代表开始符号*；
③ 小写字母的名字，如 expr 和 stmt。

- 文法符号，即非终结符或终结符：$X\ Y\ Z...$
- 终结符号串：$u\ v\ w...$
- 文法符号串：$\alpha\ \beta...$

4.2 上下文无关文法

- **符号表示约定**
  - 不同可选体: A→ α_1 , A→ α_2 , A→ α_3...可写作： A→$\alpha_1|\alpha_2|\alpha_3$...
- **有了上面的这些约定，可以直接用P代替四元组来描述G**
  - 约定：第一个产生式左部的符号是文法开始符号。

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id}$$

4.2 上下文无关文法

哪些是终结符，哪些是非终结符？

$$
\begin{aligned}
\textit{expression} &\rightarrow \textit{expression} + \textit{term} \\
\textit{expression} &\rightarrow \textit{expression} - \textit{term} \\
\textit{expression} &\rightarrow \textit{term} \\
\textit{term} &\rightarrow \textit{term} * \textit{factor} \\
\textit{term} &\rightarrow \textit{term} \;/\; \textit{factor} \\
\textit{term} &\rightarrow \textit{factor} \\
\textit{factor} &\rightarrow (\; \textit{expression} \;) \\
\textit{factor} &\rightarrow \mathbf{id}
\end{aligned}
$$

4.2 上下文无关文法

产生式，又可称为重写规则
（Rewriting rule）

$$
\begin{aligned}
expression &\rightarrow expression + term \\
expression &\rightarrow expression - term \\
expression &\rightarrow term \\
term &\rightarrow term * factor \\
term &\rightarrow term \,/\, factor \\
term &\rightarrow factor \\
factor &\rightarrow (\; expression \;) \\
factor &\rightarrow \mathbf{id}
\end{aligned}
$$

推导：右部 替换 左部

从开始符号出发，不断进行替换，就可以得到文法不同句型

递归定义注意出口

4.2 上下文无关文法

- **推导的实例1**
  - 从 expression 推导出 id/id*id

    expression => term

    => term*factor

    => term/factor*factor

    => factor/factor*factor

    =>id/factor*factor

    =>id/id*facto

    =>id/id*id

expression	→	*expression* + *term*
expression	→	*expression* - *term*
expression	→	*term*
term	→	*term* * *factor*
term	→	*term* / *factor*
term	→	*factor*
factor	→	(*expression*)
factor	→	**id**

4.2 上下文无关文法

- **推导的实例2**
  - E → E+E | E*E | -E | (E) | **id**, 从上述文法推导出串 -(id+id)
    - *E* => - *E*
    - ⇒ -(*E*)
    - ⇒ -(*E* + *E*)
    - ⇒ -(id + *E*)
    - ⇒ -(id + id)

4.2 上下文无关文法

- **直接推导($\Rightarrow$)**
  - 如果$A\rightarrow\gamma$是一个产生式，则有$\alpha A\beta\Rightarrow\alpha\gamma\beta$，其中$\Rightarrow$表示一步推导(用$A\rightarrow\gamma$)，这时称$\alpha\gamma\beta$是由$\alpha A\beta$直接推导的。
    - “$\Rightarrow$” 的含义是，使用一条规则，代替$\Rightarrow$左边的某个符号，产生$\Rightarrow$右端的符号串
- **推导($\Rightarrow^+$)**
  - 若通过一步或多步推导可从α串导出β串，则称β为α的推导，并记为$\alpha\Rightarrow^+\beta$
- **星推导($\Rightarrow^*$)**
  - 称$\alpha\Rightarrow^*\beta$ 当且仅当$\alpha=\beta$ 或 $\alpha\Rightarrow^+\beta$

4.2 上下文无关文法

- **句型**
  - 如果有$S \Rightarrow^* \alpha$, S是文法的开始符, $\alpha \in (V_T \cup V_N)^*$，则称$\alpha$是G的句型
- **句子**
  - 如果有$S \Rightarrow^+ \alpha$, S是文法的开始符, $\alpha \in V_T^*$，则称α是G的句子
- **语言**
  - $L(G) = \{\alpha \mid S \Rightarrow^+ \alpha, \alpha \in V_T^*\}$， 即文法G的所有句子的集合
- **文法等价**
  - G1和G2两个文法,若$L(G1) = L(G2)$,则称G1和G2等价

此处回想一下FA接受的字符串

4.2 上下文无关文法

- **实例1**
  - 从 expression 推导出 id/id*id

expression => term

=> term*factor

=> term/factor*factor

=> factor/factor*factor

=>id/factor*factor

=>id/id*facto

=>id/id*id

句型

句子

expression	→	*expression* + *term*
expression	→	*expression* - *term*
expression	→	*term*
term	→	*term* * *factor*
term	→	*term* / *factor*
term	→	*factor*
factor	→	(*expression*)
factor	→	**id**

4.2 上下文无关文法

- **实例2**

P:
(1) E → T
(2) E → E + T
(3) T → F
(4) T → T * F
(5) F → (E)
(6) F → i
(7) F → n

句型: E, T, E+T, F, T*F, i, n, (E), …….

句子: i, n, (i), (n), i+i, i+n, ……

语言:{i, n, (i), (n), i+i, i+n , ……}

注意：以上所有概念都不局限于上下文无关文法，而适用于所有文法(e.g. 正规文法)

4.2 上下文无关文法

- **从推导的角度看，语法分析的任务是**
  - 接受一个终结符号串作为输入，找出从文法的开始符号推导出这个串的方法

4.2 上下文无关文法

- **推导中可能遇到的两个问题**

expression	→	*expression* + *term*
expression	→	*expression* - *term*
expression	→	*term*
term	→	*term* * *factor*
term	→	*term* / *factor*
term	→	*factor*
factor	→	(*expression*)
factor	→	**id**

expression => term
=> term*factor
=> term/factor*factor

- Q1: 每一步替换哪个非终结符号？
- Q2: 若以某个非终结符号为头的产生式有多个，用哪个产生式的右部替换？-留待LL分析时讲解(关键字: 向前一步预测，消除左递归，提左因子)

4.2 上下文无关文法

- **Solution to Q1:**
  - 最左(右)推导：在推导过程中，总是对当前符号串中最左(右)边的非终极符进行替换，称为最左(右)推导,记为$\Rightarrow_{l(r)m}$
  - 规范推导：最右推导
  - 左(右)句型：通过最左(右)推导得到的句型 $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$ 称为左(右)句型
  - 规范句型：右句型

4.2 上下文无关文法

P:
(1) E → T
(2) E → E + T
(3) T → F
(4) T → T * F
(5) F → (E)
(6) F → i
(7) F → n

- 最左推导:

  $i+T*n+F \Rightarrow_{lm} i+F*n+F$

- 最右推导:

  $i+T*n+F \Rightarrow_{rm} i+T*n+i$

- 左句型: i + F*n + F
- 右句型: i + T*n +i

4.2 上下文无关文法

- **从推导的角度看，语法分析的任务是**
  - 接受一个终结符号串作为输入，找出从文法的开始符号推导出这个串的方法
  - 辅助技术：语法分析树/解析树

4.2 上下文无关文法

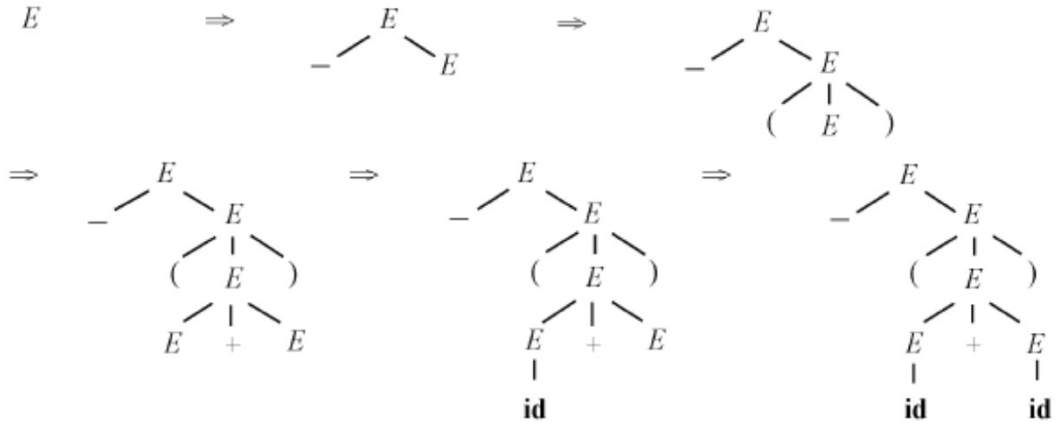
- **解析树(Parsing Tree), 又称语法分析树(syntax analysis tree)**
  - 分析树是推导的图形表示
  - 每个内部结点由非终结符标记，其子结点由该非终结符的这次推导所用产生式的右部各符号从左到右依次标记
  - 叶结点既可以是非终结符也可以是终结符，
  - 所有这些标记从左到右构成一个句型, 成为这棵树的结果(yield)或边缘(frontier)

4.2 上下文无关文法

- **由推导构造分析树**
  - 假设有推导序列：
    - $A => a_1 => a_2 => \ldots => a_n$
  - 算法：
    - 初始化：a_1的分析树是标号为A的单个结点
    - 假设已经构造出$a_{i-1}=X_1X_2\ldots X_k$的分析树，且a_{i-1}到a_1的推导是将X_j替换为β，那么在当前分析树中找出第j个非ε结点，向这个结点增加构成β的子结点。如果β=ε，则增加一个标号为ε的子结点）

4.2 上下文无关文法

- **例子：从E → E+E | E*E | -E | (E) | id, 推导出串 -(id+id)**
  - 最左推导：*E* => - *E* => -(*E*)=>-(*E* + *E*)=>-(id + *E*)=>-(id + id)



-(id + id)最左推导和最右推导的最终的分析树是一样的，也就是分析树忽略了不同的推导次序。

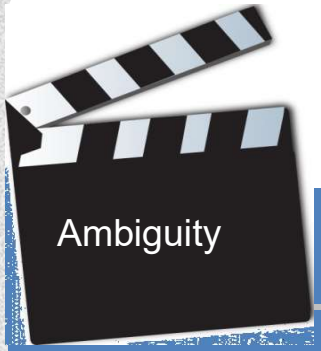
每棵分析树都和唯一的最左推导或最右推导相关联。

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

4.2 上下文无关文法

- **验证文法生成的语言**
  - 验证文法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：
    - 证明$L(G)\subseteq L$：G生成的每个串都在L中 -按照推导序列长度进行数学归纳
    - 证明$L\subseteq L(G)$：L的每个串都能由G生成 -按照符号串的长度来构造推导序列
    - 一般使用数学归纳法
  - 实例：文法G：$S \rightarrow (S)S|\varepsilon$, L(G)={所有具有对称括号对的串}

    (Page 129-例4.12)



4.3 二义性

4.3 二义性

- **如果一个文法可以为一个句子生成多棵不同的语法分析树，则该文法为二义性文法**

E→E+E | E*E | -E | (E) | **id**

给定一个CFG，判断它是否是二义，是不可计算的。

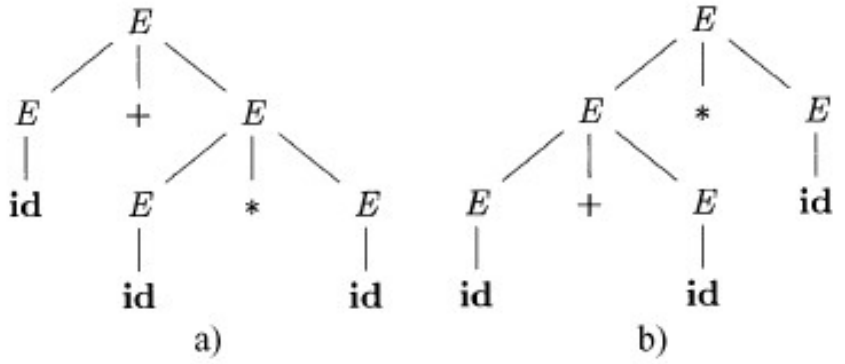


图 4-5　**id** + **id** * **id** 的两棵语法树

4.3 二义性

- **二义性会给语法分析带来不确定性，程序设计语言的文法通常都应该是无二义性的，否则就会导致一个程序有多种“正确”的解释。**
  - 应尽量避免写出二义性的文法。但有时用二义性文法定义语法结构更简单直观。
- **文法二义性问题是不可判定的**
  - 不存在一种算法，能在有限步内确切的判定一个文法不是二义性文法。
  - 若要证明文法是二义性的，只要举出一个有两棵以上语法分析树的句子即可。

4.3 二义性

- **通常解决方案**
  - 消除文法中的二义性
  - 控制文法的二义性，即加入人为的附加条件，来剔除不要的语法分析树(更常用)
    - 则二义文法的存在并非坏事，有时二义性文法可以简化文法的表示。
  - 常用策略
    - 规定符号的优先级
    - 规定符号的结合性

4.3 二义性

- **消除二义性**

有二义性的表达式文法:

Exp → Exp + Exp
Exp → Exp – Exp
Exp → Exp * Exp
Exp → Exp / Exp
Exp → (Exp)
Exp → id
Exp → num

无二义性的表达式的文法:

E → T
E → E + T | E - T
T → F
T → T * F | T / F
F → (E)
F → id
F → num

4.3 二义性

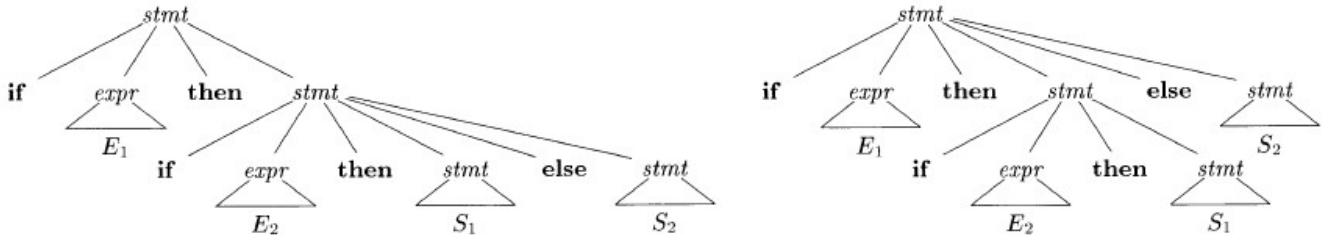
- **消除二义性**

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 的两棵语法树



4.3 二义性

- **消除二义性**
  - 问题出在哪？
    - else stmt 和第几个then配对？
    - 通常约定：和最近的尚未匹配的then配对
  - 如何在文法上体现该约定？
    - 在一个then和else之间出现的语句必须是“已匹配的”，也就是说，中间的语句不能以一个尚未匹配的(或者说开放的)then结尾
    - 一个已匹配的语句要么是一个不包含开放语句的if-then-else的完整结构；要么是一个非条件语句(other).

4.3 二义性

- **消除二义性**
  - 改写文法：引入新的非终结符号matched_stmt，用来区分是否是成对的then/else

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

$$
\begin{aligned}
stmt &\rightarrow matched\_stmt \\
&\mid open\_stmt \\
matched\_stmt &\rightarrow \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } matched\_stmt \\
&\mid \textbf{other} \\
open\_stmt &\rightarrow \textbf{if } expr \textbf{ then } stmt \\
&\mid \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } open\_stmt
\end{aligned}
$$

4.3 二义性

- 二义性的消除方法没有规律可循
- 通常并不是通过改变文法来消除二义性

4.4 CFG v.s. FSG

4.4 CFG v.s. FSG

- **每个正则语言都是一个上下文无关语言，反之不成立**
  - RE (a|b)*abb的产生式描述

$$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$$
$$A_1 \rightarrow bA_2$$
$$A_2 \rightarrow bA_3$$
$$A_3 \rightarrow \epsilon$$

  - 可以机械地将一个NFA用CFG描述

1）对于 NFA 的每个状态 i，创建一个非终结符号 A_i。

2）如果状态 i 有一个在输入 a 上到达状态 j 的转换，则加入产生式 $A_i \rightarrow aA_j$。如果状态 i 在输入 ϵ 上到达状态 j，则加入产生式 $A_i \rightarrow A_j$。

3）如果 i 是一个接受状态，则加入产生式 $A_i \rightarrow \epsilon$。

4）如果 i 是自动机的开始状态，令 A_i 为所得文法的开始符号。

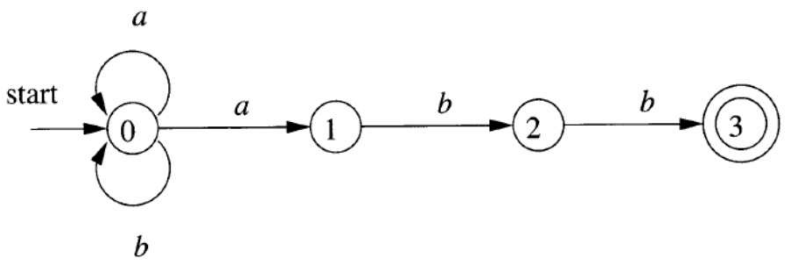
4.4 CFG v.s. FSG

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

$A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow \varepsilon$



考虑baabb的推导和接受过程

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

4.4 CFG v.s. FSG

- **CFG S→aSb | ab 描述了语言{$a^n b^n$|n>0}，但是这个语言无法用RE描述，证明如下：**
  - 假设有DFA识别此语言，且这个文法有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，$a^j b^j$必然到达接受状态，因此$a^i b^j$必然也到达接受状态
  - 通俗地说：有穷自动机不能计数，因为它不能记录下在它看到第一个b之前读入的a的个数

4.4 CFG v.s. FSG

- **思考：对于语言$\{a^n b^j c^i \mid n \geq 0, j \geq 0, 0 \leq i \leq n\}$**
  - 能用正则文法描述吗？
  - 描述它的文法是怎样的？

4.4 CFG v.s. FSG

- **分离词法分析和语法分析**
  - Q1：既然正规集都是上下文无关语言，那么为什么要用正规式定义语言的词法？其理由如下：
    - (1) 语言的词法规则非常简单，不必用功能更强的上下文无关文法描述它。
    - (2) 对于词法记号，正规式给出的描述比上下文无关文法给出的描述更简洁且易于理解。
    - (3) 从正规式自动构造出的词法分析器比从上下文无关文法构造出的更有效。

4.4 CFG v.s. FSG

- **分离词法分析和语法分析**
  - Q2: 为什么不把词法分析并入到语法分析中，直接从字符流进行语法分析，即把语言字母表上的字母作为语法分析的终结符
    - (1) 简化设计是最重要的考虑
      - 如果词法分析和语法分析合在一起，必须将语言的注解和空白的规则反映在文法中，这将使文法大大复杂。注解和空白由自己来处理的分析器，比注解和空格已由词法分析器删除的分析器要复杂得多。

4.4 CFG v.s. FSG

- **分离词法分析和语法分析**
  - Q2: 为什么不把词法分析并入到语法分析中，直接从字符流进行语法分析，即把语言字母表上的字母作为语法分析的终结符
    - (2) 编译器的效率会改进
      - 词法分析的分离可以简化词法分析器的设计，允许构造专门的和更有效的词法分析器。编译的相当一部分时间消耗在读源程序和把它分成一个个记号上，专门的读字符和处理记号的技术可以加快编译速度。

4.4 CFG v.s. FSG

- **分离词法分析和语法分析**
  - Q2: 为什么不把词法分析并入到语法分析中，直接从字符流进行语法分析，即把语言字母表上的字母作为语法分析的终结符
    - (3) 编译器的可移植性加强
      - 输入字符集的特殊性和其他与设备有关的不规则性可以限制在词法分析器中，特殊的或非标准的符号的表示，如Ｐａｓｃａｌ的↑，可以分离在词法分析器中处理。

4.4 CFG v.s. FSG

- **分离词法分析和语法分析**
  - Q2: 为什么不把词法分析并入到语法分析中，直接从字符流进行语法分析，即把语言字母表上的字母作为语法分析的终结符
    - (4) 把语言的语法结构分成词法和非词法两部分，为编译器前端的模块划分提供了方便的途径。

4.4 CFG v.s. FSG

- **分离词法分析和语法分析**
  - 哪些应作为词法规则，哪些应作为语法规则，没有严格的准则。正规式是描述诸如标识符、常数和关键字等词法结构的最有力武器。
  - 上下文无关文法是描述括号配对、begin和end配对、语句嵌套、表达式嵌套等结构的最有力武器，这些结构不可能用正规式来描述。

4.4 CFG v.s. FSG

- **CFG局限性**
  - 可以对2个个体进行计数a^nb^n，但无法对三个个体进行计数$a^nb^nc^n$
  - 无法描述L={wcw|w=(a|b)*}
    - 它抽象地表示了C或者Java中“标识符先声明后使用”的规则
    - 说明了C/Java这些语言不是上下文无关语言，不能使用上下文无关文法描述。
    - 通常用上下文无关文法描述其基本结构，不能用文法描述的特性在语义分析阶段完成。原因是上下文文法具有高效的处理算法。

4.4 CFG v.s. FSG

- **文法设计**
  - 程序设计语言所需要的文法
    - 上下文无关文法足够用来描述语法吗？
      - 标识符必须先声明后使用
    - 语法分析器能够完全按照上下文无关文法来构造吗？
      - 二义性、左递归的文法可能给语法分析器造成很大麻烦
  - 可以怎么做？
    - 改造语法分析器，添加语义规则，使它可以做得更多
    - 改造上下文无关文法，消除二义性和左递归，提取左因子

4.5 语法错误处理

4.5 语法错误处理

- 编译器的任务是检测**invalid**的程序，并将**valid**的程序翻译成目标代码。

Error Kind	Example	Detected by
词法错误	...@...	scanner
语法错误	...x*%...	parser
语义错误	int x;y=x(3);	type checker

4.5 语法错误处理

- 错误处理器应该做到：
  - 及时、准确、清晰地报告错误；
  - 能够从错误中迅速地恢复；
  - 不要降低valid程序的编译速度；
- **Good error handling is not easy to achieve!**

4.5 语法错误处理

- 语法错误恢复的方法
  - 由简单到复杂
  - **Panic mode**
  - **Error productions**
  - **Automatic local or global correction**

Panic mode

- 最简单，最经常使用的方法
- 当检测到一个语法错误时，
- 丢掉一些**token**，直到遇到一些具有清晰分隔作用的符号；
- 然后从该符号处开始分析。
- 这些符号称为“同步符号”，一般地为语句或表达式的结束符。

Panic mode的例子

- 对于下面错误的表达式 (1++2)+3 Panic mode 恢复：
  - 当语法分析遇到第二个“+”时，跳到下一整数2处继续分析。
  - 这种恢复方式可以通过在文法中增加一个特殊的终极符error实现：

    E → int | E + E | (E) | error int | (error)

Error productions

- 思想：将一些常见的错误写进产生式
- **Example: 5*x　写成　5x**
- 增加产生式　**E → …… | EE**
- 不利：使文法变得复杂了。

Local and Global Correction

- 思想：找到一个“最接近的”正确的程序
- 需要插入或删除一些**token**; 穷举搜索
- 不利：
  - 难于实现；
  - 减慢正确程序的分析；
  - “最接近的”不一定是需要的程序；
  - 有些自动工具不支持。

作业

- CFG
  - 教材P130: 4.2.1, 4.2.2(6), 4.2.3(2)(5), 4.2.5
  - 教材P137: 4.3.3

Thank you!