第2章 文法与语言

主要内容

- 程序设计语言
- 语言概述
- 形式语言
- 符号和符号串
- 文法
- 语言（推导）
- 语法树与句型分析
- 文法二义性
- 文法与语言分类

主要内容

- 程序设计语言
- 语言概述
- 形式语言
- 符号和符号串
- 文法
- 语言（推导）
- 语法树与句型分析
- 文法二义性
- 文法与语言分类

程序设计语言范型

- 命令式语言
- 函数式语言
- 基于规则的语言

PROLOG

Hanoi(N):-move(N,left,centre,right)

Move(0,-,-,-):-!

Move(N,A,B,C):-M is N-1,
move(M,A,C,B),move(M,C,B,A)

?-Hanoi(3)

LISP:

Define (function1) (paras) (statements)
(function2) (paras) (statements)
(function3) (paras) (statements)
…
(functionn) (paras) (statements)

(functioni actual-paras)

(functioni actual-paras)

(functioni actual-paras)

…

主要内容

- 程序设计语言
- 语言概述
- 形式语言
- 符号和符号串
- 文法
- 语言（推导）
- 语法树与句型分析
- 文法二义性
- 文法与语言分类

语言与文法

- **当表述一种语言时**，无非是说明这种语言的句子，如果语言只含有穷多个句子，则只需**列出句子的有穷集**就行了；但对于含有无穷句子的语言，就出现**如何给出它的有穷表示**的问题
- 以自然语言为例，人们无法列出全部句子，但是人们可以给出一些规则，**用这些规则来说明（或者定义）句子的组成结构。**比如，汉语句子可以是由主语后随谓语而成，构成谓语的是动词和直接宾语，可以采用**EBNF**来表示这种句子的构成规则

“我是大学生”是汉语的一个句子

〈句子〉::=〈主语〉〈谓语〉

〈主语〉::=〈代词〉|〈名词〉

〈代词〉::=<u>我</u>|<u>你</u>|<u>他</u>

〈名词〉::=<u>王明</u>|<u>大学生</u>|<u>工人</u>|<u>英语</u>

〈谓语〉::=〈动词〉〈直接宾语〉

〈动词〉::=<u>是</u>|<u>学习</u>

〈直接宾语〉::=〈代词〉|〈名词〉

有了一组规则以后，按照如下方式用它们导出句子：

开始去找::=左端的带有〈句子〉的规则并把它由::=右端的符号串代替，这个动作表示成：

〈句子〉 ⇒ 〈主语〉〈谓语〉，

然后，在得到串〈主语〉〈谓语〉中，选取〈主语〉或〈谓语〉，再用相应规则的::=右端代替之。比如，选取了〈主语〉，并采用规则〈主语〉::=〈代词〉，

那么得到：〈主语〉〈谓语〉 ⇒ 〈代词〉〈谓语〉，

重复做下去，

句子："我是大学生"的全部动作过程是：

〈句子〉 ⇒ 〈主语〉〈谓语〉 ⇒ 〈代词〉〈谓语〉

⇒ 我〈谓语〉 ⇒ 我〈动词〉〈直接宾语〉

⇒ 我是〈直接宾语〉 ⇒ 我是〈名词〉 ⇒ 我是大学生

“我是大学生”的构成符合上述规则，是句子

“我大学生是”不符合上述规则，不是句子

这些规则成为判别句子结构合法与否的依据，换句话说，这些规则可以看成是一种元语言，用它描述汉语

这里仅仅涉及汉语句子的结构描述，其中一种描述元语言称为文法

英语句子

sentence –> **<subject> <verb-phrase> <object>**

subject –> This | Computers | I

verb-phrase –> **<adverb> <verb>** | **<verb>**

adverb –> never

verb –> is | run | am | tell

object –> the **<noun>** | a **<noun> | <noun>**

noun –> university | world | cheese | lies

This is a university.

Computers run the world.

I am the cheese.

I never tell lies.

语言概述

语言是其句子的集合

汉语--所有符合汉语语法的句子的全体

英语--所有符合英语语法的句子的全体

程序设计语言--所有该语言的程序的全体

研究语言：
- 每个句子构成的规律
- 每个句子的含义
- 每个句子和使用者的关系

研究程序设计语言

- 每个程序构成的规律
- 每个程序的含义
- 每个程序和使用者的关系

语言研究的三个方面

- 语法　Syntax
- 语义　Semantics
- 语用　Pragmatics

语法 -- 表示构成语言句子的各个记号之间的组合规律

语义 -- 表示各个记号及其组合的特定含义（各个记号和记号所表示对象之间的关系）

语用 -- 表示在各个记号所出现的行为中，它们的来源、使用和影响 (**Merriam-Webster**: A branch of linguistics that is concerned with the relationship of sentences to the environment in which they occur; **Stanford Encyclopedia**: Pragmatics deals with utterances, by which we will mean specific events, the intentional acts of speakers at times and places, typically involving language)

每种语言具有两个可识别的特性：**语言的形式**和**与该形式相关联的意义**

语言的实例若在语法上是正确的，与其**相关联的意义**可以从两个观点来看：其一是该句子的创立者所想要表示的意义，另一是接收者所检验到的意义。这两个意义并非总是一样的，前者称为语言的语义，后者是其语用意义。**幽默**、**双关语**和**谜语**就是利用这两方面意义间的差异。

讨论：语义和语用不一致，如果出现在高级程序设计语言中呢？

如果不考虑语义和语用，即只从**语法这一侧面**来看语言，这种意义下的语言称作**形式语言**

形式语言抽象地定义为一个数学系统，“**形式**”是指这样的事实：语言的所有规则<u>只以什么符号串能出现</u>的方式来陈述

主要内容

- 程序设计语言
- 语言概述
- 形式语言
- 符号和符号串
- 文法
- 语言（推导）
- 语法树与句型分析
- 文法二义性
- 文法与语言分类

形式语言

- 如语言学中语言一样，形式语言一般有两个方面: 语法和语义。专门研究语言语法的数学和计算机科学分支叫做**形式语言理论**，它只研究语言的语法而不是它的语义。
- 在形式语言理论中，**形式语言**是一个字母表上的某些有限长字符串的集合。一个形式语言可以包含无限多个字符串。
- 形式语言理论是对符号串集合的**表示法**、**结构**及其**特性**的研究，是**程序设计语言语法分析研究的基础**

Formal language

- In automata theory, a *formal language* is a set of strings of symbols drawn from a finite *alphabet*. A formal language can be specified either by a set of rules (such as regular expressions or a context-free grammar) that generates the language, or by a *formal machine* that *accepts*(*recognizes*) the language. A formal machine takes strings of symbols as input and outputs either “yes” or “no”. A machine is said to accept a language if it says “yes” to all and only those strings that are in the language. Alternatively, a language can be defined as the set of strings for which a particular machine says “yes” .

Definition of formal language

Basic definitions:

- A ***symbol*** is our basic building block, typically a character or a digit.
- An ***alphabet*** is a finite set of symbols.
- A ***string*** is a finite sequence of alphabet symbols.
- A *formal language* is a set of strings (possibly infinite), all over the same alphabet.

主要内容

- 程序设计语言
- 语言概述
- 形式语言
- 符号和符号串
- 文法
- 语言（推导）
- 语法树与句型分析
- 文法二义性
- 文法与语言分类

符号和符号串

- 任何一种语言都是由该语言的基本符号组成的符号串集合

符号和符号串简介

一些基本概念：

字母表

符号

符号串（空符号串）

符号串集合

- **符号** 一个抽象实体,我们不再形式地定义它(就象几何中的”点”一样).例如字母是符号,数字也是符号。
- **字母表** 字母表是元素的非空有穷集合，我们把字母表中的元素称为符号，因此字母表也称为**符号集。**不同的语言可以有不同的字母表，**例如**：汉语的字母表中包括汉字、数字及标点符号等，PASCAL语言的字母表是由字母、数字、若干专用符号及BEGIN、IF之类的保留字组成。

符号串 由字母表中的符号组成的任何有穷序列称为符号串。

例如， 00 11 10 是字母表Σ =｛0, 1｝上的符号串。字母表A =｛a,b,c｝上的一些符号串有：a, b, c, ab, aaca。

在符号串中，符号的顺序是很重要的，符号串ab就不同于ba，abca和aabc也不同。

可以使用字母表示符号串，如x=STR表示“x是由符号S、T和R，并按此顺序组成的符号串”。

符号串的长度 如果某符号串x中有m个符号，则称其长度为m，表示为｜x｜= m，如001110的长度是6。

空符号串 即不包含任何符号的符号串，用 ε 表示，其长度为0，即｜ε｜=0。

符号串的头、尾，固有头和固有尾：如果z=xy是一符号串，那么x是z的头，y是z的尾，如果x是非空的，那么y是固有尾；如果y非空，那么x是固有头。

例如：设z=abc，那么z的头是**ε**, a, ab, abc，除abc外，其它都是固有头；z的尾是**ε**, c, bc, abc, z的固有尾是**ε**, c, bc。

当对符号串z=xy的头感兴趣而对其余部分不感兴趣时，采用**省略写法**：z=x…；

如果**为了强调x在符号串z中的某处出现**，则可表示为：z=…x…；符号t是符号串z的第一个符号，则表示为z=t…。

符号串的连接：设x和y是符号串，它们的连接xy是把y的符号写在x的符号之后得到的符号串. 由于ε的含义，显然有ε x=x，x ε =x

例如：x=ST，y=abu，则它们的连接xy=STabu，看出｜x｜=2，｜y｜=3，｜xy｜=5

符号串的方幂：符号串自身连接n次得到的符号串，a^n 定义为 aa…aa n个a，a^1=a, a^2=aa且a^0=ε

例如：若x=AB，则:

$x^0 = \varepsilon$

x^1 =AB

x^2 = ABAB

x^3 = ABABAB

$x^n = xx^{n-1} = x^{n-1}\,x \quad (n>0)$

符号串集合：若集合A中所有元素都是某字母表Σ上的符号串，则称A为字母表Σ上的符号串集合。

两个符号串集合A和B的乘积：

定义为 $AB = \{xy \mid x \in A 且 y \in B\}$

若集合 $A=\{ab,cde\}$ 集合 $B = \{0,1\}$，则

$AB = \{ab1, ab0, cde0, cde1\}$

使用 Σ^* 表示Σ上的一切符号串（包括ε）组成的集合。**Σ^*称为Σ的闭包。**

Σ上的除ε外的所有符号串组成的集合记为Σ^+。**Σ^+称为Σ的正闭包**。

主要内容

- 程序设计语言
- 语言概述
- 形式语言
- 符号和符号串
- 文法
- 语言（推导）
- 语法树与句型分析
- 文法二义性
- 文法与语言分类

文法

- 符号→符号串→句子→语言
- 并非所有符号串都能形成句子

如何描述一种语言？

如果语言是有穷的（只含有有穷多个句子），可以将句子逐一列出来表示

如果语言是无穷的，找出语言的有穷表示。语言的有穷表示有两个途经：

生成方式（**文法**）：语言中的每个句子可以用严格定义的规则来构造。

识别方式（**自动机**）：用一个过程，当输入的一任意串属于语言时，该过程经有限次计算后就会停止并回答"是"，若不属于，要么能停止并回答"不是"，要么永远继续下去。

- 文法即是生成方式描述语言的：语言中的每个句子可以用一组严格定义的规则来构造。
- 以下将给出文法的定义，进而在文法的定义的基础上，给出推导的概念，句型、句子和语言的定义。

文法定义

- 文法是对语言结构的形式化定义和描述。
- 文法由规则（产生式）构成。
- 规则：U::=x 或U → x
- 文法G[S]，开始符号，字汇表/字母表，终结符号，非终结符

回顾**EBNF** 引入的符号(元符号)：

< >　用左右尖括号括起来的语法成分为非终结符

∷=　(→)　‘定义为’　∷=(→) 的左部由右部定义

|　　‘或’

{ }　　表示花括号内的语法成分可重复任意次或限定次数

[ ]　　表示方括号内的语法成分为任选项

( )　　表示圆括号内的成分优先

例：用**EBNF**描述〈整数〉的定义：

〈整数〉::=[+|-]〈数字〉{〈数字〉}

〈数字〉::=0|1|2|3|4|5|6|7|8|9

或更好的写法：

〈整数〉::=[+|-]〈非零数字〉{〈数字〉}|0

〈非零数字〉::=1|2|3|4|5|6|7|8|9

〈数字〉::=0|〈非零数字〉

文法G定义为一个四元组(V_N，V_T，P，S)，其中：

V_N为非终结符号(或语法实体，或变量)集；

V_T为终结符号集；

P为产生式(也称规则)的集合；V_N，V_T和P是非空有穷集。

S称作识别符号或开始符号，它是一个非终结符，至少要在一条产生式中作为左部出现。

V_N和V_T不含公共的元素，即$V_N \cap V_T = \phi$

用V表示$V_N \cup V_T$，称为文法G的字母表或字汇表

规则，也称**重写规则**、**产生式**或**生成式**，是形如$\alpha \rightarrow \beta$或$\alpha ::= \beta$的(α，β)有序对，其中α是字母表V的正闭包V^+中的一个符号，β是V^*中的一个符号。α称为规则的左部，β称作规则的右部。

文法举例

- 例：G[E]:

  E ::=E+T

  E ::=T

  T ::=T*F

  T ::=F

  F ::=(E)

  F ::=i

  V_N =? V_T =? V=?

- $\mathbf{V_N}$ = {E,T,F}
- $\mathbf{V_T}$ ={+,(,),*,i}
- $\mathbf{V}$={E,T,F,+,(,),*,i}

Define a Grammar

A grammar G is defined as a 4-tuple（V_N，V_T，P，S）

V_N is a set of ***nonterminals***

V_T is a set of ***terminals***

P is a set of ***productions***, each production consists of a ***left side***, an arrow(or '::='), and a ***right side***

S is a designation of one of the nonterminals as the ***start symbol***

$V = V_N \cup V_T$ is the alphabet of G

文法的定义

例 文法G=（V_N，V_T，P，S）

V_N = { S }, V_T ={ 0, 1 }

P={ S→0S1, S→01 }

S为开始符号

例 文法G=（V_N，V_T，P，S）

V_N ={标识符，字母，数字}

V_T ={a,b,c,…x,y,z,0,1,…,9}

P={ <标识符>→<字母>

<标识符>→<标识符><字母>

<标识符>→<标识符><数字>

<字母>→a,…, <字母>→z

<数字>→0, …, <数字>→9 }

S=<标识符>

文法的写法

1 G：S→aAb

A→ab

A→aAb

A→ε

2 G[S]： A→ab A→aAb **A→ε S→aA**b

3 G[S]： A→ab |aAb **|ε S→aA**b

元符号：→

∷=

|

< >

习惯表示:

大写字母：非终结符

小写字母：终结符

S –> AB

A –> Ax | y

B –> z

递归

- 递归
- 递归规则，递归文法
- 左递归，右递归，直接递归，间接递归

主要内容

- 程序设计语言
- 语言概述
- 形式语言
- 符号和符号串
- 文法
- 语言（推导）
- 语法树与句型分析
- 文法二义性
- 文法与语言分类

推导的定义

直接推导“$\Rightarrow$”

$\alpha\rightarrow\beta$是文法G的产生式，若有δ_1, δ_2满足： $\delta_1=\gamma_1\alpha\gamma_2$, $\delta_2=\gamma_1\beta\gamma_2$, 其中$\gamma_1,\gamma_2\in V^*$

则称δ_1直接*推导*到δ_2,记作 $\delta_1 \Rightarrow \delta_2$

也称δ_2直接*归约*到δ_1

例：G： $S\rightarrow 0S1$， $S\rightarrow 01$

$0S1 \Rightarrow 00S11$

$00S11 \Rightarrow 000S111$

$000S111 \Rightarrow 00001111$

$S \Rightarrow 0S1$

<程序>⇒<分程序>.

<分程序>. ⇒ <变量说明部分> <语句>.

VAR<标识符>;BEGIN READ(<标识符>) END.

⇒

VAR A;BEGIN READ(<标识符>) END.

推导的定义

若存在$\delta = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \dots \Rightarrow \gamma_n = \gamma$, (n>0)
则记为$\delta \Rightarrow^+ \gamma$，$\delta$推导出$\gamma$，或$\gamma$归约到$\delta$

若有$\delta \Rightarrow^+ \gamma$，或$\delta = \gamma$，则记为$\delta \Rightarrow^* \gamma$

例：G： $S \rightarrow 0S1$， $S \rightarrow 01$

$0S1 \Rightarrow 00S11$

$00S11 \Rightarrow 000S111$

$000S111 \Rightarrow 00001111$

$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 00001111$

$S \Rightarrow^{+} 00001111$

$S \Rightarrow^{*} S$ $00S11 \Rightarrow^{*} 00S11$

What are Derivations

Derivation is a way that a grammar defines a language. In the process of derivation a production is treated as a rewriting rule in which the nonterminal on the left side is replaced by the string on the right side of the production.

A production $u \to v$ is used by replacing an occurrence of u by v. Formally, if we apply a production p of $\boldsymbol{P}$ to a string of symbols w in $\boldsymbol{V}$ to yield a new string of symbols z in $\boldsymbol{V}$, we say that z derived from w using p, written as follows: $w \Rightarrow^{p} z$. We also use:

$w \Rightarrow z$ z derives from w (production unspecified)

$w \Rightarrow^{*} z$ z derives from w using zero or more productions

$w \Rightarrow^{+} z$ z derives from w using one or more productions

在上下文无关文法中：$\mathbf{A} \rightarrow \beta \ (\alpha \in V_N)$

- **最左推导：** $x\mathbf{A}\delta \Rightarrow x\boldsymbol{\alpha}\delta,\ x \in V_t^*$
- **最右推导：** $\gamma\mathbf{A}y \Rightarrow \gamma\boldsymbol{\alpha}y,\ y \in V_t^*$

- **规范推导：** 最右推导
- 规范归约：最左归约

- 例：G[E]:

  E::=E+T

  E::= T

  T ::=F

  T ::=T*F

  F ::=(E)

  F ::=a

- a+a*a的推导:

句型、句子的定义

句型：

有文法G，若S =>* x，则称x是文法G的句型。

句子：

有文法G，若S =>* x，且x∈V_T^*，则称x是文法G的句子。

例：G： S→0S1， S→01

S ⇒0S1 ⇒00S11 ⇒000S111 ⇒00001111

G的句型S, 0S1, 00S11,000S111, 00001111

G的句子00001111, 01

右句型

- 由最右推导（规范推导）所得的句型称为**右句型**（规范句型）

例：G[E]： $E \to E+T|T$

$T \to T*F|F$

$F \to (E)|a$

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

句子：用符号**a，+，*，(，**和 **)** 构成的算术表达式

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

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

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

语言的定义

由文法G生成的语言记为L(G),它是文法G的一切句子的集合：

$L(G)=\{x|S \Rightarrow^* x$，其中S为文法的开始符号，且$x \in V_T^*\}$

例：G：　$S \rightarrow 0S1$，　$S \rightarrow 01$

$L(G)=\{0^n1^n|n \geqslant 1\}$

例： 文法G[S]：

（1）S→aSBE

（2）S→aBE

（3）EB→BE

（4）aB→ab

（5）bB→bb

（6）bE→be

（7）eE→ee

$$L(G)=\{ a^n b^n e^n \mid n \geqslant 1 \}$$

$$
\begin{aligned}
S &\Rightarrow a\,S\,BE && (S \rightarrow aSBE) \\
&\Rightarrow a\,aBEBE && (S \rightarrow aBE) \\
&\Rightarrow aabEBE && (aB \rightarrow ab) \\
&\Rightarrow aabBEE && (EB \rightarrow BE) \\
&\Rightarrow aabbEE && (bB \rightarrow bb) \\
&\Rightarrow aabbeE && (bE \rightarrow be) \\
&\Rightarrow aabbee && (eE \rightarrow ee)
\end{aligned}
$$

G生成的每个串都在L(G)中

L(G)中的每个串确实能被G生成

使用产生式(1)n-1次，得到推导序列：

$S \Rightarrow^* a^{n-1}S(BE)^{n-1}$，然后使用产生式(2)一次，得到：

$S \Rightarrow^* a^{n-1}S(BE)^{n-1} \Rightarrow a^n(BE)^n$。然后从$a^n(BE)^n$继续推导，总是对EB使用产生式(3)的右部进行替换，而最终在得到的串中，所有的B都先于所有的E。例如，若n=3,

aaaBEBEBE ⇒ aaaBBEEBE ⇒ aaaBBEBEE ⇒ aaaBBBEEE。

即有：$S \Rightarrow^* a^nB^nE^n$

接着，使用产生式(4)一次，得到$S \Rightarrow^* a^nbB^{n-1}E^n$，然后使用产生式(5)n-1次得到：

$S \Rightarrow^* a^nb^nE^n$，最后使用产生式(6)一次，使用产生式(7)n-1次，得到：$S \Rightarrow^* a^nb^ne^n$

也能证明，

对于$n \geq 1$，串$a^nb^ne^n$是唯一形式的终结符号串

文法等价

若$L(G_1)=L(G_2)$, 则称文法G_1和G_2是等价的。

如文法$G_1[A]$：$A \rightarrow 0R$ 与$G_2[S]$：$S \rightarrow 0S1$ 等价

$A \rightarrow 01$　　　　　　$S \rightarrow 01$

$R \rightarrow A1$

主要内容

- 程序设计语言
- 语言概述
- 形式语言
- 符号和符号串
- 文法
- 语言（推导）
- 语法树与句型分析
- 文法二义性
- 文法与语言分类

语法树

文法G[Z]的语法树：

- 每个结点都是G的符号
- 树根是文法的开始符号
- 若某个结点至少有一个从它出来的分支，则该结点一定是非终结符
- 若某个结点A有n个分支，假设其分支结点为B1,B2,…Bn，则

  A::=B1B2B3…Bn

  一定是文法的一条规则

语法树

- 语法树可以从推导过程产生

- 凡使用一条规则推导，则可以从规则左部符号结点长出若干分支

构造语法树

G[E]:

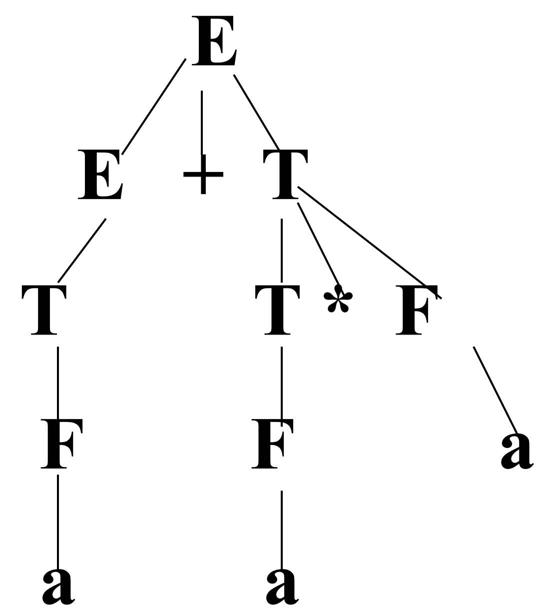
$$E \rightarrow E+T|T$$
$$T \rightarrow T*F|F$$
$$F \rightarrow (E)|a$$

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

E⇒E+T ⇒T+T ⇒F+T
⇒a+T ⇒a+T*F
⇒a+F*F ⇒a+a*F
⇒a+a*a

E⇒E+T ⇒E+T*F ⇒E+T*a
⇒E+F*a ⇒E+a*a
⇒T+a*a ⇒F+a*a
⇒a+a*a

E⇒E+T ⇒T+T ⇒T+T*F
⇒F+T*F ⇒F+F*F
⇒a+F*F ⇒a+F*a
⇒a+a*a



看不出句型中的符号被替代的顺序

句型分析

*句型分析*就是识别一个符号串是否为某文法的句型，是某个推导的构造过程

在语言的编译实现中，把完成句型分析的程序称为*分析程序*或*识别程序*，分析算法又称*识别算法*

从左到右的分析算法，即总是从左到右地识别输入符号串，首先识别符号串中的最左符号，进而依次识别右边的一个符号，直到分析结束

句型分析算法分类

分析算法可分为：

自上而下分析法：

从文法的开始符号出发，反复使用文法的产生式，寻找与输入符号串匹配的推导。

自下而上分析法：

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。

两种方法反映了两种语法树的构造过程

*自上而下方法*是从文法符号开始，将它做为语法树的根，向下逐步建立语法树，使语法树的结果正好是输入符号串

*自下而上方法*则是从输入符号串开始，以它做为语法树的结果，自底向上地构造语法树

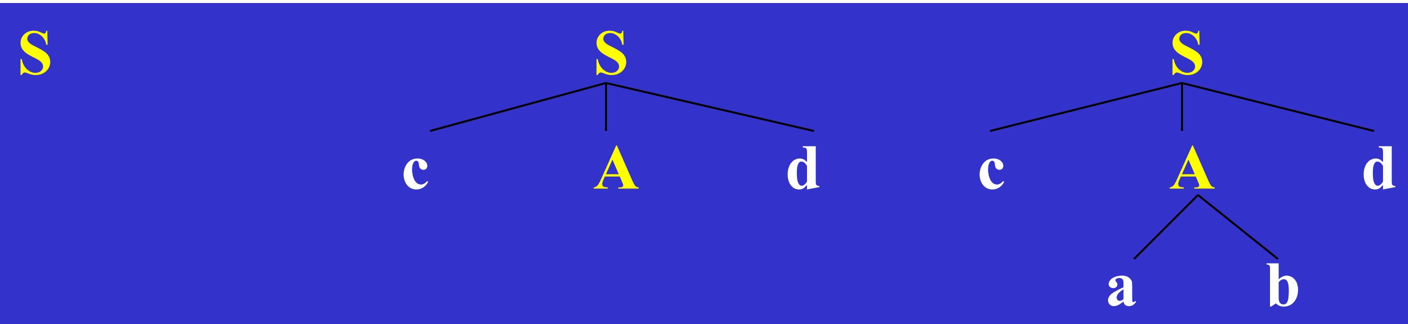
自上而下的语法分析

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



推导过程：$S \Rightarrow cAd$　　　$cAd \Rightarrow c\underline{ab}d$

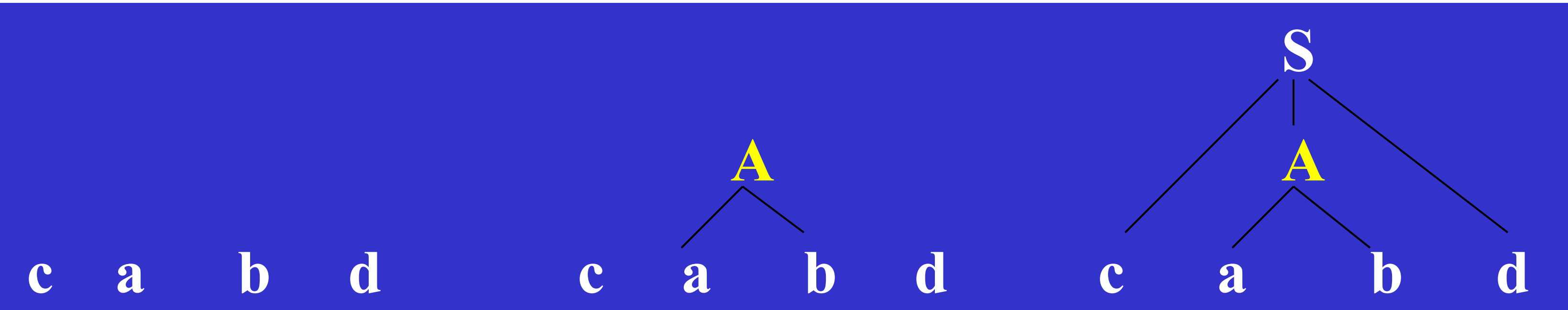
自下而上的语法分析

例：文法G： $S \rightarrow cAd$

$A \rightarrow ab$

$A \rightarrow a$

识别输入串w=cabd是否该文法的句子



归约过程构造的推导： $cAd \Rightarrow c\underline{ab}d$ $S \Rightarrow cAd$

(1) $S \rightarrow$ **cAd** (2) $A \rightarrow$ **ab** (3) $A \rightarrow$ **a**

识别输入串w=cabd是否为该文法的句子

自上而下的语法分析

若$S \Rightarrow cAd$ 后选择(3)扩展A，

$S \Rightarrow cAd \Rightarrow cad$

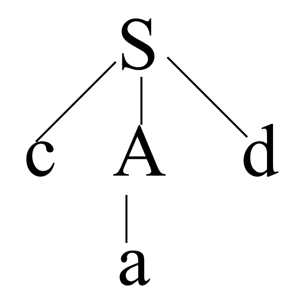
那将会？

w的第二个符号可以与叶子结点a得以匹配，但第三个符号却不能与下一叶子结点d匹配?

宣告分析失败（其意味着，识别程序不能为串cad构造语法树，即cad不是句子）

-显然是错误的结论。

导致失败的原因是在分析中对A的选择不是正确的。



(1) S → cAd (2) A → ab (3) A → a

识别输入串w=cabd是否为该文法的句子

自下而上的语法分析

对串cabd的分析中，如果不是选择ab用产生式(2),而是选择a用产生式(3)将a归约到了A，那么最终就达不到归约到S的结果，因而也无从知道cabd是一个句子

c a b d

c A b d

a

句型分析的有关问题

1）在自上而下的分析方法中如何选择使用哪个产生式进行推导？

假定要被代换的最左非终结符号是A，且有n条规则：A→B1|B2|…|Bn，那么如何确定用哪个右部去替代A？

2）在自下而上的分析方法中如何识别可归约的串？

在分析程序工作的每一步，都是从当前串中选择一个子串，将它归约到某个非终结符号，该子串称为“可归约串”

刻画“可归约串”

文法G[S]

句型的短语

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

句型的直接短语

若有 $A \Rightarrow \beta$，则称 β 是句型 $\alpha\beta\delta$ 相对于非终结符A 的直接短语

句型的句柄

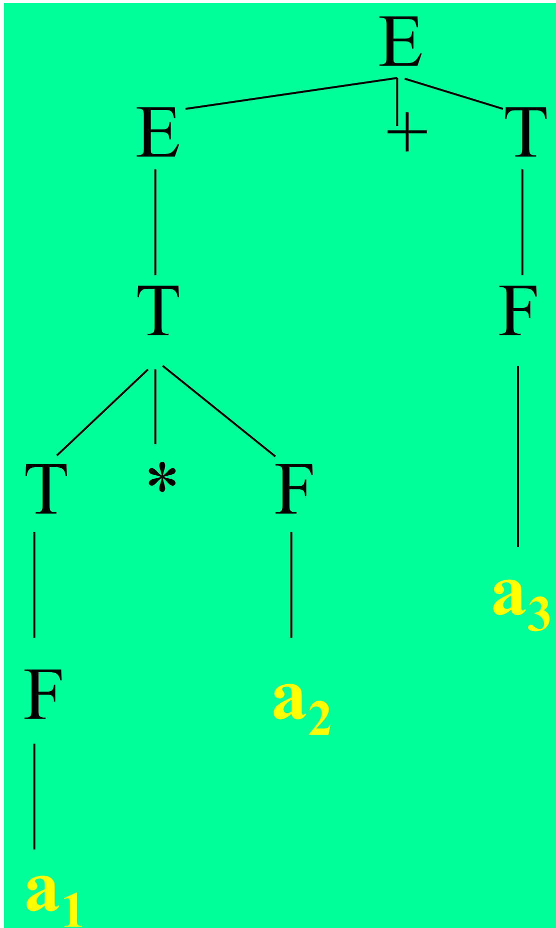
一个右句型的最左直接短语称为该句型的句柄

（~~一个句型的最左直接短语称为该句型的句柄~~）

（~~一个右句型的直接短语称为该句型的句柄~~）

- **短语**是句型中某非终结符号通过若干步**推导**出的子串
- **归约**：如果每次都从当前句型的**句柄**进行归约，则可以归约到文法的开始符号

例 ：a*a+a 的短语、直接短语和句柄



G[E]： $E \rightarrow E+T|T$

$T \rightarrow T*F|F$

$F \rightarrow (E)|a$

句型： a*a+a

短语： $a_1 * a_2 + a_3$， $a_1 * a_2$，

a_1， a_2， a_3

直接短语： a_1， a_2， a_3

句柄： a_1

从语法树判断句型的短语、直接短语和句柄

– 子树
– 简单子树
– 短语
– 直接短语
– 句柄

主要内容

- 程序设计语言
- 语言概述
- 形式语言
- 符号和符号串
- 文法
- 语言（推导）
- 语法树与句型分析
- 文法二义性
- 文法与语言分类

文法二义性

- 文法二义性：两棵语法树对应同一句子
- 根据语法树，可以发现文法的二义性

二义文法

若一个文法存在某个句子对应两棵不同的语法树，则称这个文法是二义的
或者，若一个文法存在某个句子有两个不同的最左（右）推导，则称这个文法是二义的

判定任给的一个上下文无关文法是否二义，或它是否产生一个先天二义的上下文无关语言，**这两个问题是递归不可解的**，但可以为无二义性寻找一组充分条件

文法的二义性和语言的二义性是两个不同的概念：可能有两个不同的文法G和G′，其中G是二义的，但是却有L(G)=L(G′)，即，这两个文法所产生的语言是相同的。

二义文法改造为无二义文法：

$G[E]: E \to i$	$G[E]: E \to T \mid E+T$
$E \to E+E$	$T \to F \mid T*F$
$E \to E*E$	$F \to (E) \mid i$
$E \to (E)$	*规定优先顺序和结合律*

如果产生上下文无关语言的每一个文法都是二义的，则说此语言是先天二义的。对于一个程序设计语言来说，常常希望它的文法是无二义的，因为希望对它每个语句的分析是唯一的。

主要内容

- 程序设计语言
- 语言概述
- 形式语言
- 符号和符号串
- 文法
- 语言（推导）
- 语法树与句型分析
- 文法二义性
- 文法与语言分类

文法和语言

- 文法→语言
- 语言→文法

文法的类型

通过对产生式施加不同的限制，**Chomsky**将文法分为四种类型：

- **0**型文法：对任一产生式$\alpha \to \beta$，都有$\alpha \in (V_N \cup V_T)^+$，$\beta \in (V_N \cup V_T)^*$
- **1**型文法：对任一产生式$\alpha \to \beta$，都有$|\beta| \geq |\alpha|$，仅 $S \to \varepsilon$ 除外，**S**为文法初始符号且不出现在任何产生式右边
- **2**型文法：对任一产生式$\alpha \to \beta$，都有$\alpha \in V_N$，$\beta \in (V_N \cup V_T)^*$
- **3**型文法：任一产生式$\alpha \to \beta$的形式都为$A \to aB$或$A \to a$，其中$A \in V_N$，$B \in V_N$，$a \in V_T$

A hierarchy of grammars

Type 0: free or unrestricted grammars

These are the most general. Productions are of the form *u* –> *v* where both *u* and *v* are arbitrary strings of symbols in ***V***, with u non-null. There are no restrictions on what appears on the left or right-hand side other than the left-hand side must be non-empty.

Type 1: context-sensitive grammars

Productions are of the form *uXw* –> *uvw* where *u* , *v* and w are arbitrary strings of symbols in ***V***, with *v* non-null, and *X* a single nonterminal. In other words, *X* may be replaced by *v* but only when it is surrounded by *u* and *w*. (i.e. in a particular context).

Type 2: context-free grammars

Productions are of the form *X*–> *v* where v is an arbitrary string of symbols in ***V***, and *X* is a single nonterminal. Wherever you find *X*, you can replace with *v* (regardless of context).

Type 3: regular grammars

Productions are of the form *X*–> *a* or *X*–> *aY* where *X* and *Y* are nonterminals and *a* is a terminal. That is the left-hand side must be a single nonterminal and the right-hand side can be either a single terminal by itself or with a single nonterminal. These grammars are the most limited in terms of expressive power.

文法的类型

例：1型（上下文有关）文法

文法G[S]：

$S \rightarrow CD$

$C \rightarrow aCA$

$C \rightarrow bCB$

$AD \rightarrow aD$

$BD \rightarrow bD$

$Aa \rightarrow bD$

$Ab \rightarrow bA$

$Ba \rightarrow aB$

$Bb \rightarrow bB$

$S \rightarrow \varepsilon$

$C \rightarrow \varepsilon$

文法的类型

例：2型（上下文无关）文法

文法G[S]：
$$S \to AB$$
$$A \to BS|0$$
$$B \to SA|1$$

3型文法示例

G[S]：

$S \rightarrow 0A|1B|0$

$A \rightarrow 0A|1B|0S$

$B \rightarrow 1B|1|0$

G[I]：

$I \rightarrow lT$

$I \rightarrow l$

$T \rightarrow lT$

$T \rightarrow dT$

$T \rightarrow l$

$T \rightarrow d$

文法的实用限制

G[S]：

S→Be

B→Ce | Af

A→Ae | e

C→Cf

D→f

非终结符：

1. 不可到达的
2. 不可终止的

ε 规则

- $A \to \varepsilon$ 形式的规则的出现，使文法结论的证明变得复杂
- 但$A \to \varepsilon$ 形式的规则的引入，可让文法变得更简洁、易读

- 若L是1、2、或3型语言，则$L \cup \{\varepsilon\}$和 $L \setminus \{\varepsilon\}$ 也分别是1、2、3型语言

小结

- 程序设计语言
- 语言概述
- 形式语言
- 符号和符号串
- 文法
- 语言(推导)
- 语法树与句型分析
- 文法二义性
- 文法与语言分类

作 业

- 2;
- 5;
- 9;
- 10;

- 12(1)(3)(6);
- 15(2);
- 18;